Parveen Khurana

125 Followers About Following

Recurrent Neural Networks(RNNs)

P Parveen Khurana Jul 13, 2019 · 18 min read

This article covers the content discussed in the RNN module of the Deep Learning course offered on the website: https://padhai.onefourthlabs.in

Data and Tasks

RNNs are typically used for 3 types of **tasks**:

- **Sequence Classification** (sentiment classification, vidèo classification)
- **Sequence Labelling** (part of speech tagging, named entity recognition)
- **Sequence Generation** (machine translation, **transliteration**)

Sequence Classification: We look at the entire sequence and produce one output at the end. (for n words/video frames in the sequence we produce one output)

Sequence Labeling: For every word in the sequence we want to attach a label to that(for n words in the sequence we produce n outputs)

Named entity recognition: We identify people names, location names, organization names. Sometimes we also consider dates, numbers and so on depending on the use of the application. So, for every word we need a label whether it is NE(named entity) or

Sequence Generation:

Machine Translation: Given an input in one language we translate it to another language (given n-word input, the output could be of m words), every word could lead to more than 1 output; more than 1 input might produce a single output.

Data in case of sequence classification

x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	y
The	first	half	was	very	boring	.		0
Great	performance	by	all	the	lead	actors	.	1
The	visual	effects	were	stunning	.			1
The	movie	was	a	waste	of	time	.	0

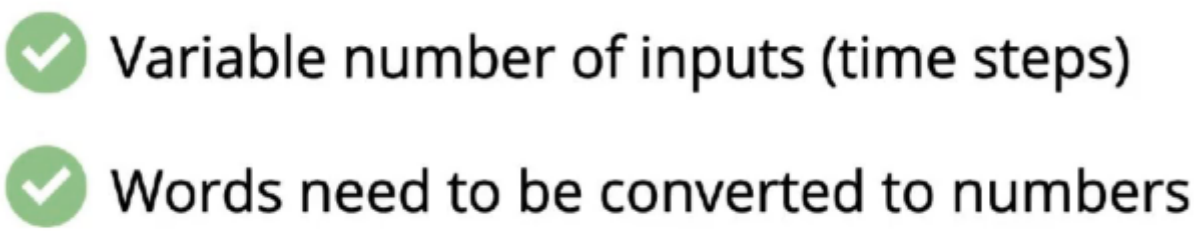
Data looks like this in the case of sequence classification

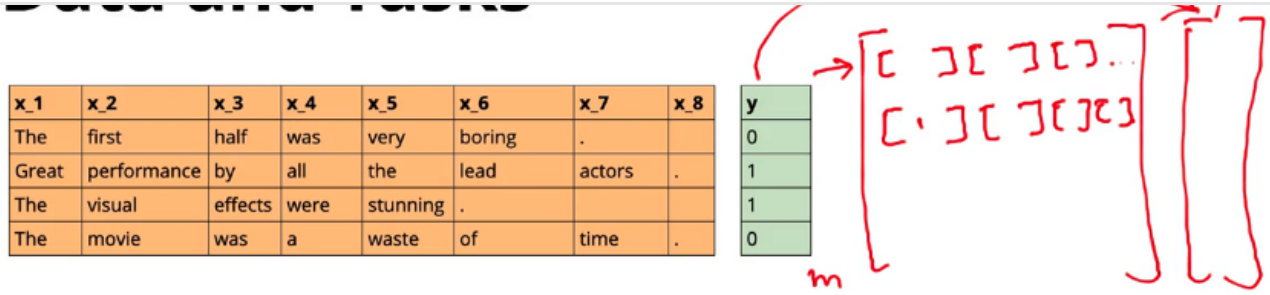
This is what the data looks like. We will be given a bunch of sentences(this is a task of sentiment classification, you could take any other task of similar nature).

Here x1, x2, x3 represents the first word, second word, third word and so on in the sentence and then the correct label is given as 'y'.

So, we would be provided with the x, y where x is a sequence of words{x1, x2, x3, ……., xn} in the sentence and y is the correct label.

So, from the above data set, it is clear that the number of words are different in each instance and the second thing is that the words need to be converted to numbers as the neural network would take numeric input.





x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	y
The	first	half	was	very	boring	.		0
Great	performance	by	all	the	lead	actors	.	1
The	visual	effects	were	stunning	.			1
The	movie	was	a	waste	of	time	.	0

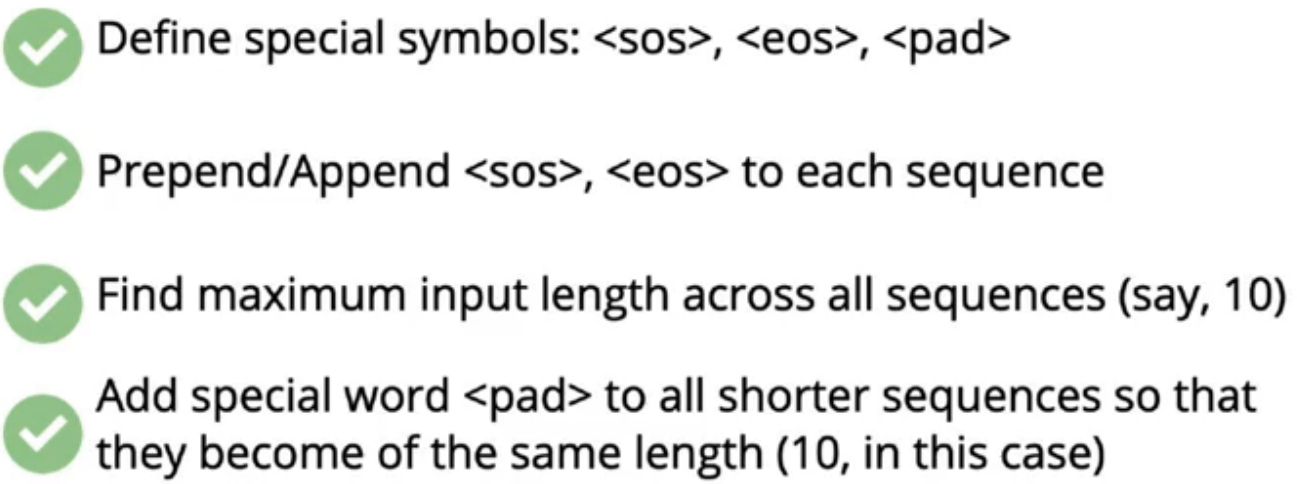
We need to represent the given data in the matrix form

We want to represent the given data in the matrix form, here each word is represented as one hot encoded vector.

The number of elements in every row is actually different(that means the dimension of each row is going to be different). Our goal is to construct a matrix X and a vector Y such that the matrix has fixed number of dimensions, it has m rows and k columns(such that for all the 'm' rows we have the same 'k' number of columns)(this would make part of data pre-processing).

Data Pre-processing before feeding it to the model:

We need to do a bunch of things in preprocessing and these are the standard for all Natural Language Processing(NLP) problems and all sequence problems:



We define special symbols as given below:

<sos> is for start of sequence: is just the first word, we insert this artificial word which just tells that this is the start of the sequence. Similarly, we have <eos> after the

<eos> is for end of sequence

We find the maximum input length across all sequences(input sentences) and then we add a special word to all the shorter sequences so that all the sentences/sequences are of the same length(After this our entire matrix would be of the same size).

x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	y
<sos>	The	first	half	was	very	boring	.	**<eos>**	**<pad>**	0
<sos>	Great	performance	by	all	the	lead	actors	.	**<eos>**	1
<sos>	The	visual	effects	were	stunning	.	**<eos>**	**<pad>**	**<pad>**	1
<sos>	The	movie	was	a	waste	of	time	.	**<eos>**	0

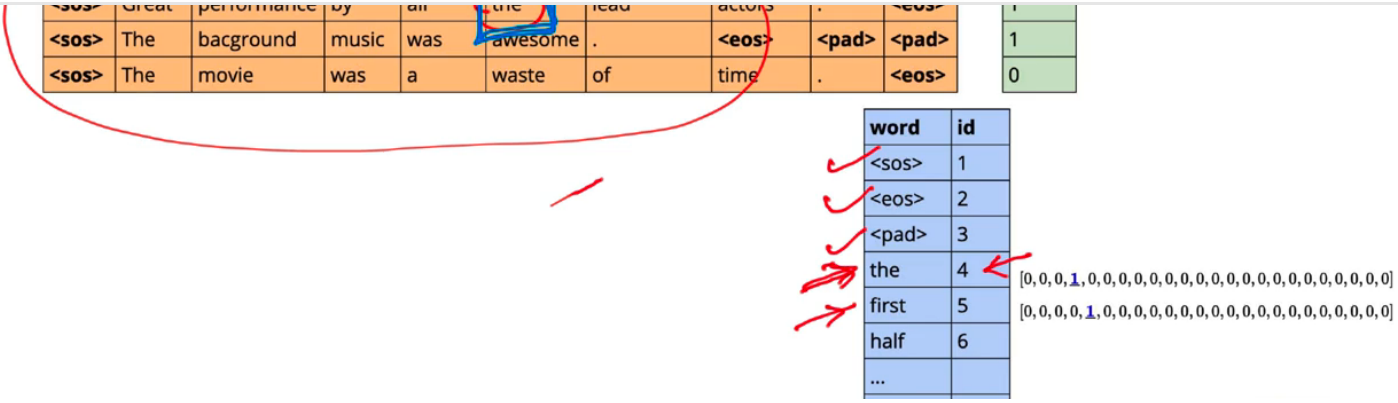
After this, we create the one hot encoded vectors for all the words/ we define the index for all the unique words that we have(in the training data), we start with the special words and then for every word that we encounter we put it in this table(if that word is not already there in the table).

word	id
<sos>	1
<eos>	2
<pad>	3
the	4
first	5
half	6
...	

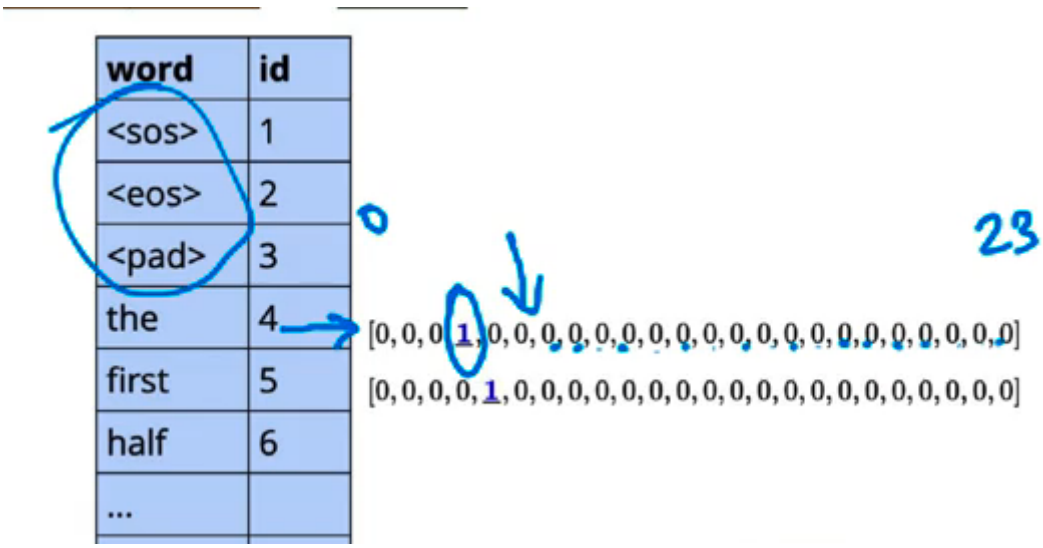
$$[0,0,0,\underline{1},0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]$$

$$[0,0,0,0,\underline{1},0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]$$

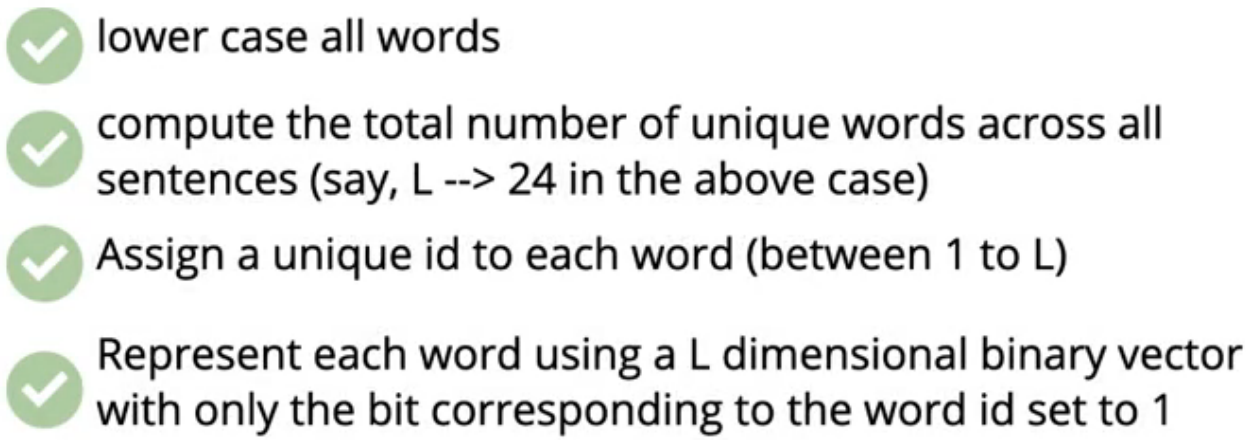
We assign id to all the special words and unique words in the training data, then this id helps to generate one hot encoded vectors as we have only one entry in that vector as 1(index of that entry is corresponding to the id assigned to the word).



So, this table would be a collection of the all unique words in the data set and for every word we have a unique index associated with it and similarly every number/index would have only one word associated with it. So, with this, we can represent every word by one hot vector(only one entry would be 1 corresponding to the word's index).



So, the complete method is:



Let's take one example:

Input data:

x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	y
<sos>	The	first	half	was	very	boring	.	<eos>	<pad>	0
<sos>	Great	performance	by	all	the	lead	actors	.	<eos>	1

We make the dictionary(collection of all the unique words, special symbols) and based on that we generate one hot encoded vector for all the words:

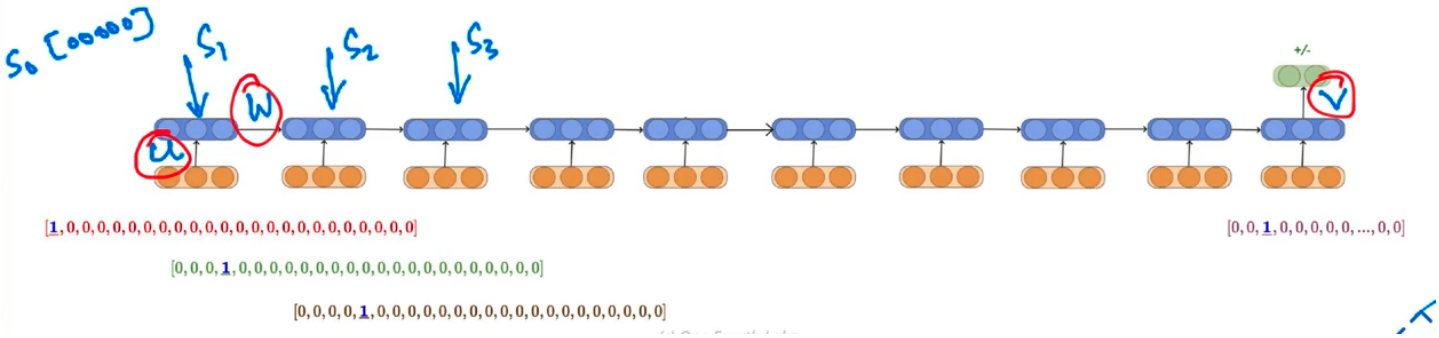
This is what one hot vectors matrix for the input data(input matrix) would look like:

We assumed that the total no. of unique words in the above case is 24(including special characters), now each of the words would be represented using 24 dimensional vector and we have 10 such words(in each sentence after the pre-processing) so we have 24*10 = 240 as the dimension/length of the each of the rows in the input data. So, if we 'm' input rows, the input matrix would be of size 'mX240'.

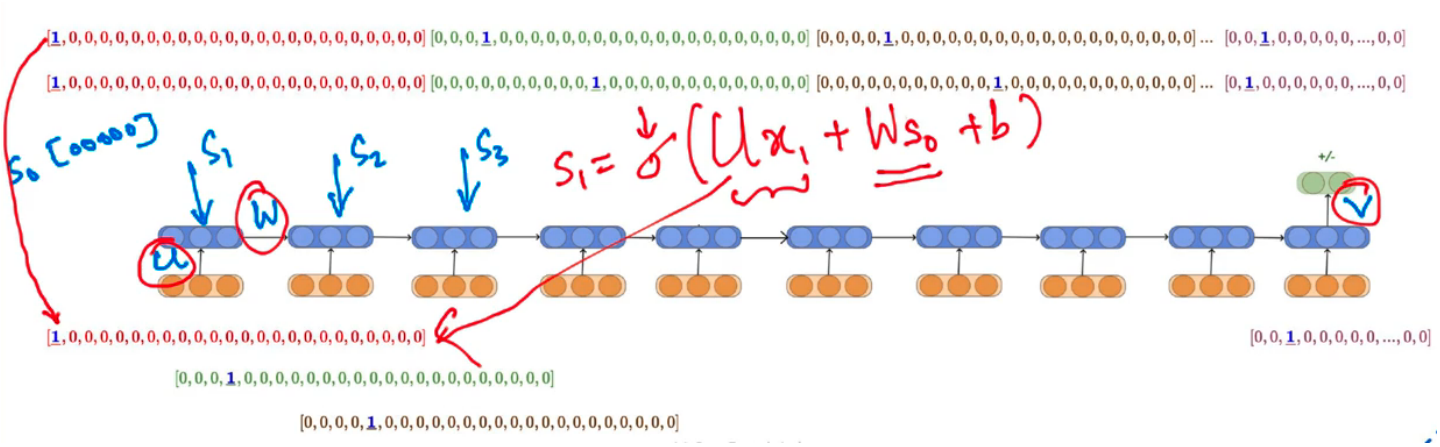
Tensorflow, PyTorch might store this in the form of indices instead of one hot vector for each word. For each word, it just stores the index(where the entry is 1 in the one hot encoder for the given word) of entry 1 in the input data matrix and it would map the index to the one hot vector under the hood. For example let' say we have two words each having a dimension of 5 and represented as [00100] [01000]; now PyTorch, Tensorflow might store this as [2 1], here 2 corresponds to the index where the entry is 1 for the first word(in one hot encoded vector), then 1 corresponds to the index where the entry is 1 for the second word(in the one hot encoded vector).

So, in this case, the matrix dimension would be 'm X L' where L is the max length.

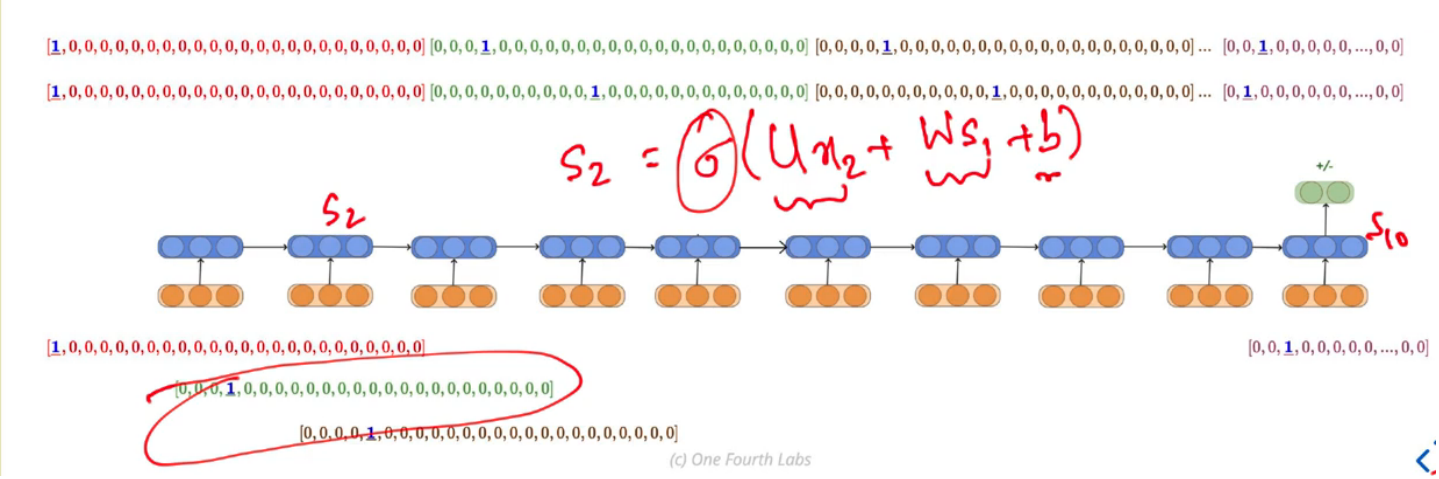
Let's say s0 is all zeroes



Now the first input(x1) would be given to the model, it would multiply it with ‘u’ and add ‘b’ to it and then on this, the non-linearity is applied.



After this, the second input from the same row comes into play



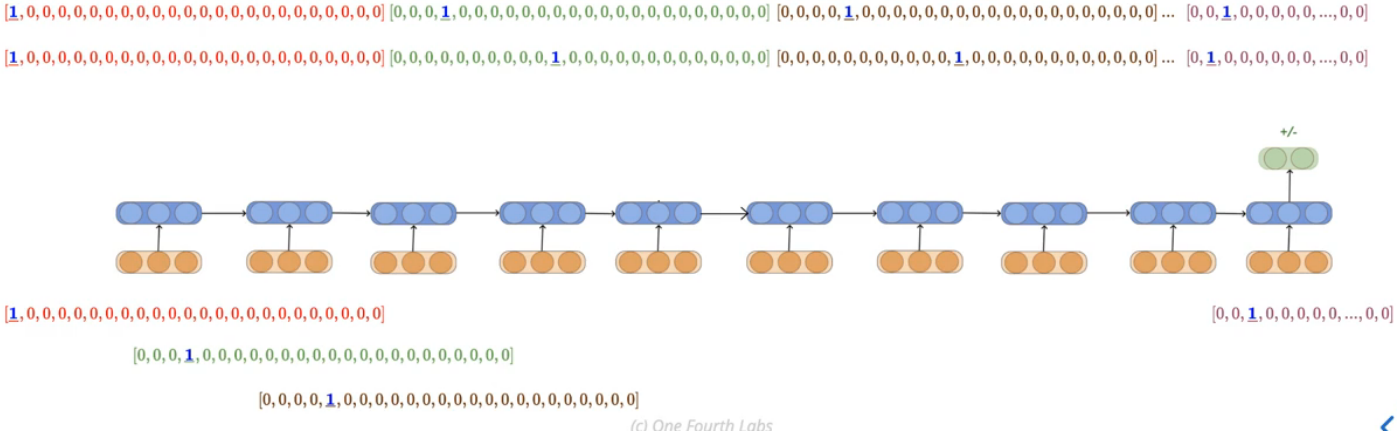
And this would be computed all the way up to s10 and then from s10 we can compute

Here O represents the softmax function.

The entire picture looks like this:

x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9
<sos>	The	first	half	was	very	boring	.	<eos>	<pad>
<sos>	Great	performance	by	all	the	lead	actors	.	<eos>

y
0
1



Padding

The question is that padding might corrupt the input as we are adding some artificial word at the end for example:

x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9
<sos>	Great	movie	<eos>	<pad>	<pad>	<pad>	<pad>	<pad>	<pad>

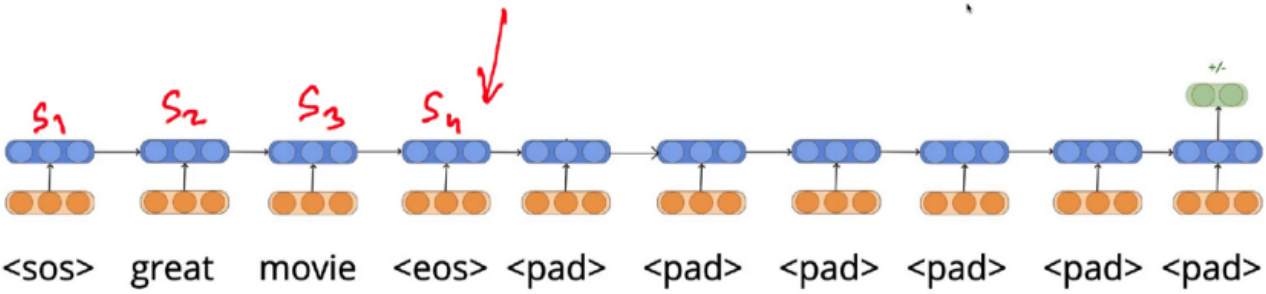
y
1

Here the original sentence(of length 2) was ‘Great movie’, we have added 6 pad,1 sos, and 1 eos to this to make it of length 10.

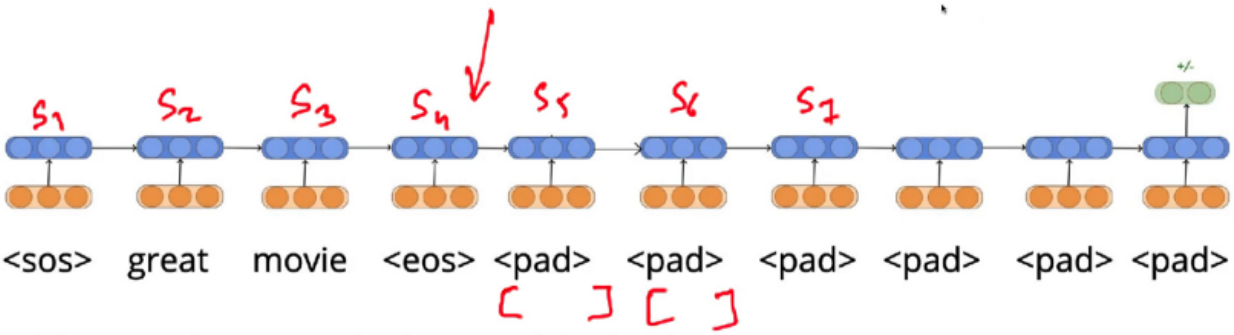
x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9

y

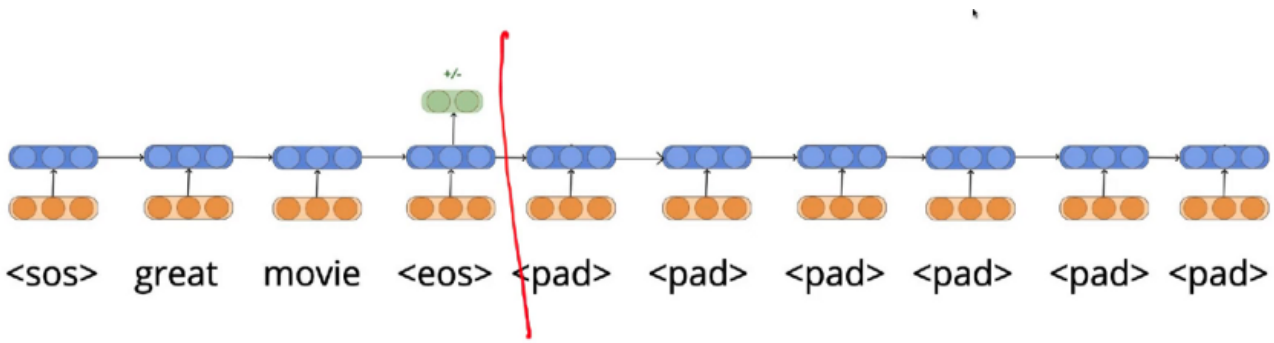
The computations for this would look like:



Ideally, the computation would have ended here after computing s4 because that's all the sentence contained but we are computing s5, s6, and so on in which case we would be feeding the pad as the input every time step(after the 4th time step) and then computing the final output at the end. This looks like it might corrupt the input.



So in practice what we do is that we take the output after the eos itself and from that we judge whether the sentence conveys positive sentiment or the negative sentiment.



padding was only to ensure that input matrix is of uniform size

In addition to this, we also feed in the Length vector containing the true length of the sentences. PyTorch, Tensorflow use it to do the computations till that point.

Data and Tasks — Sequence Labeling

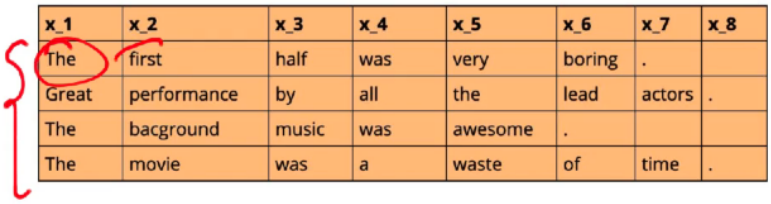
Sequence Labeling — For every word in the input sentence, we need to produce an output.

Input data would be of the form

x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8
The	first	half	was	very	boring	.	
Great	performance	by	all	the	lead	actors	.
The	bacground	music	was	awesome	.		
The	movie	was	a	waste	of	time	.

And for every word we have the corresponding output

y_1	y_2	y_3	y_4	y_5	y_6	y_7	y_8
DT	AJ	NN	VB	JJ	JJ	PC	
AJ	NN	PP	PN	DT	JJ	NN	PC
DT	JJ	BB	VB	JJ	PC		
DT	NN	VB	DT	NN	PP	NN	PC



x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8
The	first	half	was	very	boring	.	
Great	performance	by	all	the	lead	actors	.
The	bacground	music	was	awesome	.		
The	movie	was	a	waste	of	time	.

y_1	y_2	y_3	y_4	y_5	y_6	y_7	y_8
DT	AJ	NN	VB	JJ	JJ	PC	
AJ	NN	PP	PN	DT	JJ	NN	PC
DT	JJ	BB	VB	JJ	PC		
DT	NN	VB	DT	NN	PP	NN	PC

Great	performance	by	all	the	lead	actors	.
The	bacground	music	was	awesome	.		
The	movie	was	a	waste	of	time	.

AJ	NN	PP	PN	DT	JJ	NN	PC
DT	JJ	BB	VB	JJ	PC		
DT	NN	VB	DT	NN	PP	NN	PC

DT is for determiner, AJ for adjective, NN for noun, VB for verb, PC for punctuation and so on.

So, here both the inputs as well as outputs are of variable length.

- Variable number of inputs (time steps)
- Variable number of outputs (time steps)
- Words need to be converted to numbers

- Labels/outputs need to be converted to numbers

Pre-processing:

- Define special symbols/**labels**: <sos>, <eos>, <pad>
- Prepend/Append <sos>, <eos> to each input/**output** sequence

- Find maximum input length across all sequences (say, 10)
- Add special word/**label** <pad> to all shorter sequences so that they become of the same length (10, in this case)

<sos>	The	movie	was	a	waste	of	time	.	<eos>

<sos>	DT	NN	VB	DT	NN	PP	NN	PC	<eos>

x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9
<sos>	The	first	half	was	very	boring	.	<eos>	<pad>
<sos>	Great	performance	by	all	the	lead	actors	.	<eos>
<sos>	The	bacground	music	was	awesome	.	<eos>	<pad>	<pad>
<sos>	The	movie	was	a	waste	of	time	.	<eos>

y_0	y_1	y_2	y_3	y_4	y_5	y_6	y_7	y_8	y_9
<sos>	DT	AJ	NN	VB	JJ	JJ	PC	<eos>	<pad>
<so>	AJ	NN	PP	PN	DT	JJ	NN	PC	<eos>
<sos>	DT	JJ	BB	VB	JJ	PC	<eos>	<pad>	<pad>
<sos>	DT	NN	VB	DT	NN	PP	NN	PC	<eos>

x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9
<sos>	The	first	half	was	very	boring	.	<eos>	<pad>
<sos>	Great	performance	by	all	the	lead	actors	.	<eos>
<sos>	The	bacground	music	was	awesome	.	<eos>	<pad>	<pad>
<sos>	The	movie	was	a	waste	of	time	.	<eos>

y_0	y_1	y_2	y_3	y_4	y_5	y_6	y_7	y_8	y_9
<sos>	DT	AJ	NN	VB	JJ	JJ	PC	<eos>	<pad>
<so>	AJ	NN	PP	PN	DT	JJ	NN	PC	<eos>
<sos>	DT	JJ	BB	VB	JJ	PC	<eos>	<pad>	<pad>
<sos>	DT	NN	VB	DT	NN	PP	NN	PC	<eos>

Input matrix size(in this case) would be ‘m X 10’ where ‘m’ is the number of the data rows and 10(acts as indices of the words that are there in the sentence, each indice would refer to the index of the one hot vector where the entry is 1 for example let’s say the **first indice** is **1**, now this 1 implies that the **first element** of the **one hot vector would be 1** and all other elements would be 0 and since this is the **first indice** then this means it is for the **first word in the sentence**) is the maximum input length(in this case). Output matrix size would also be ‘m X 10’(here 10 acts as indices of labels that we have in the sentence).

After this we convert words to numbers:

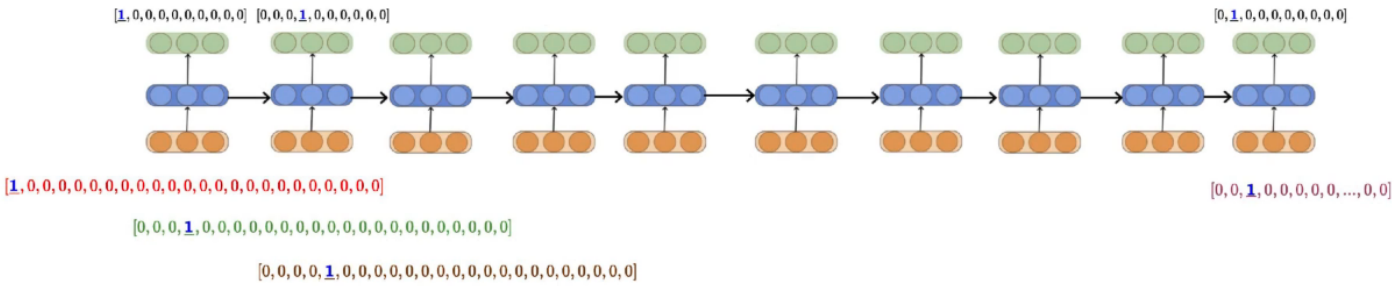
word	id
<sos>	1
<eos>	2
<pad>	3
the	4
first	5
half	6
...	
...	
time	24

$$[0,0,0,\underline{1},0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]$$
$$[0,0,0,0,\underline{1},0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]$$
$$[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,\underline{1}]$$

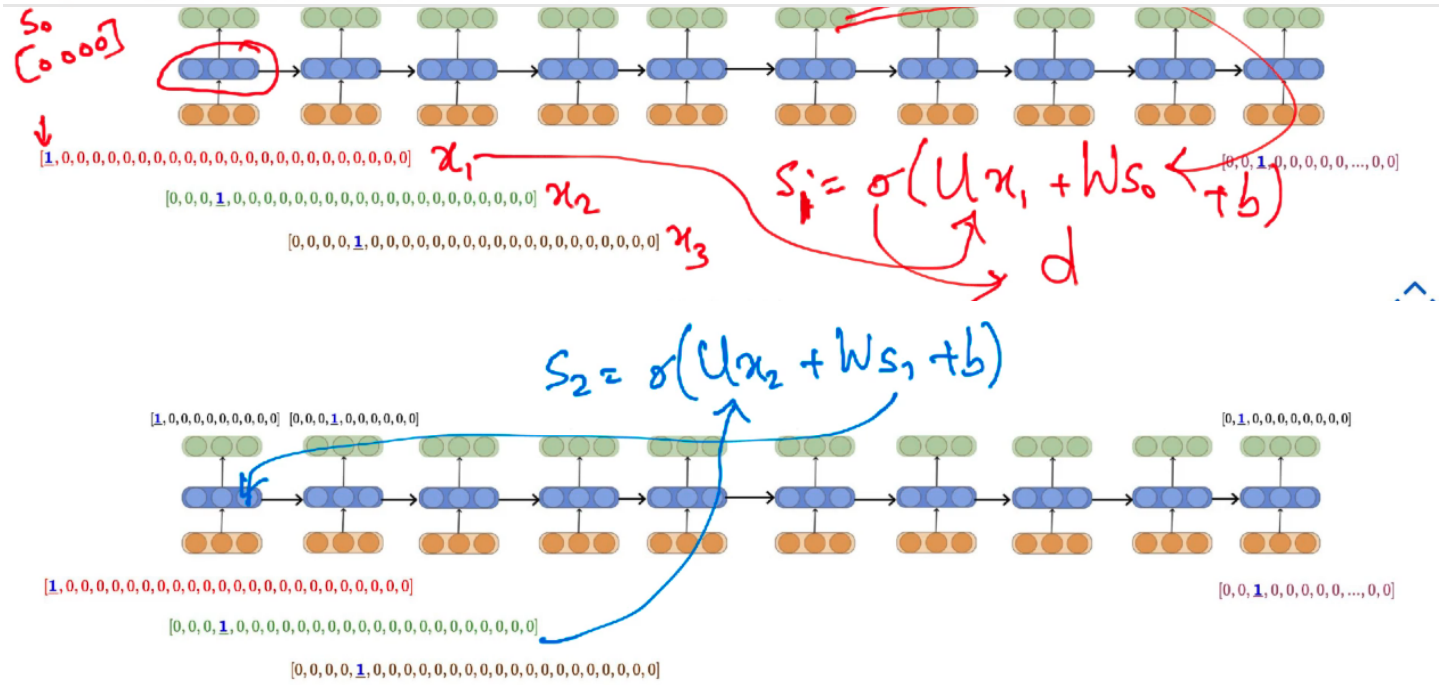
<eos>	2	
<pad>	3	
DT	4	[0, 0, 0, 1, 0, 0, 0, 0, 0, 0]
JJ	5	[0, 0, 0, 0, 1, 0, 0, 0, 0, 0]
NN	6	
...		
...		
PN	10	[0, 0, 0, 0, 0, 0, 0, 0, 0, 1]

x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9
<sos>	The	first	half	was	very	boring	.	<eos>	<pad>
<sos>	Great	performance	by	all	the	lead	actors	.	<eos>
<sos>	The	bacground	music	was	awesome	.	<eos>	<pad>	<pad>
<sos>	The	movie	was	a	waste	of	time	.	<eos>

y_0	y_1	y_2	y_3	y_4	y_5	y_6	y_7	y_8	y_9
<sos>	DT	JJ	NN	VB	JJ	JJ	PC	<eos>	<pad>
<so>	JJ	NN	PP	PN	DT	JJ	NN	PC	<eos>
<sos>	DT	JJ	NN	VB	JJ	PC	<eos>	<pad>	<pad>
<sos>	DT	NN	VB	DT	NN	PP	NN	PC	<eos>



We pass the first input(x1) of the first row to the model, it would then compute s1 using the equation in the picture below, then from this y1_hat is computed using the softmax on (Vs1 + c) and we also know the true distribution y, so we can compare the two.

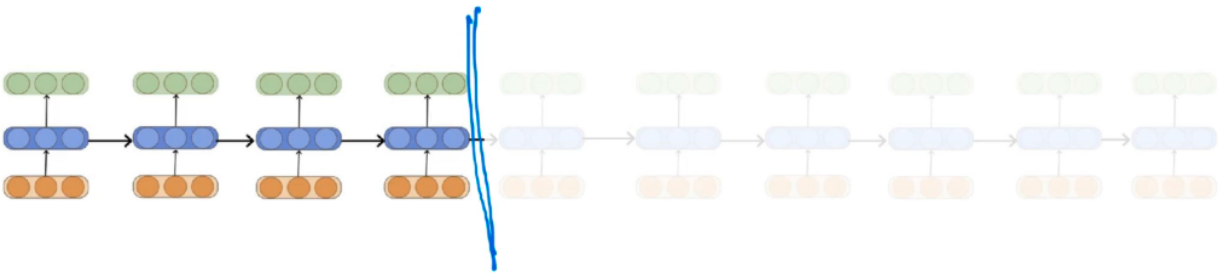


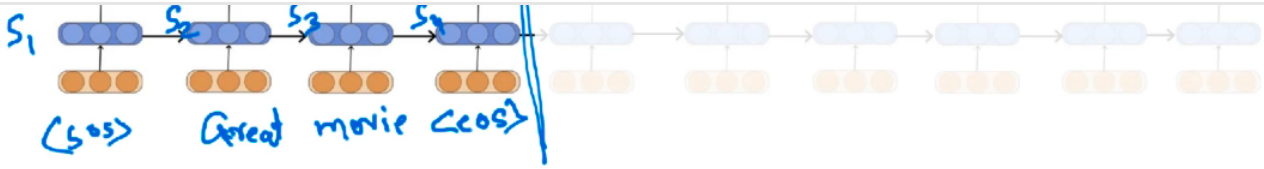
Then we pass the second element to the model and so on.

Here also the padding is just to make sure the dimensions are consistent throughout, computations would happen only up to the true length.

x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9
<sos>	Great	movie	<eos>	<pad>	<pad>	<pad>	<pad>	<pad>	<pad>

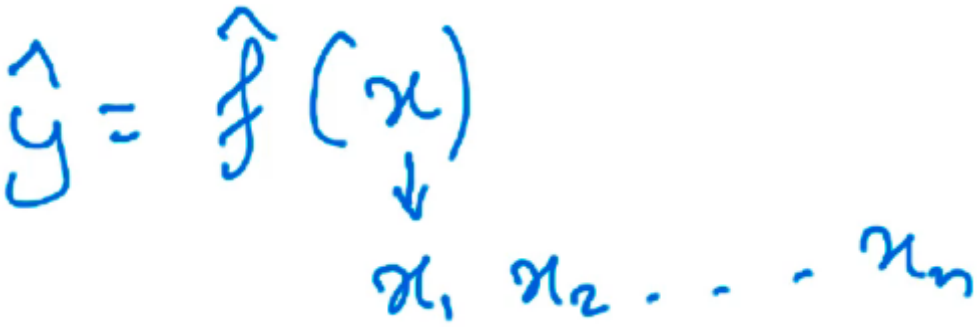
y
1





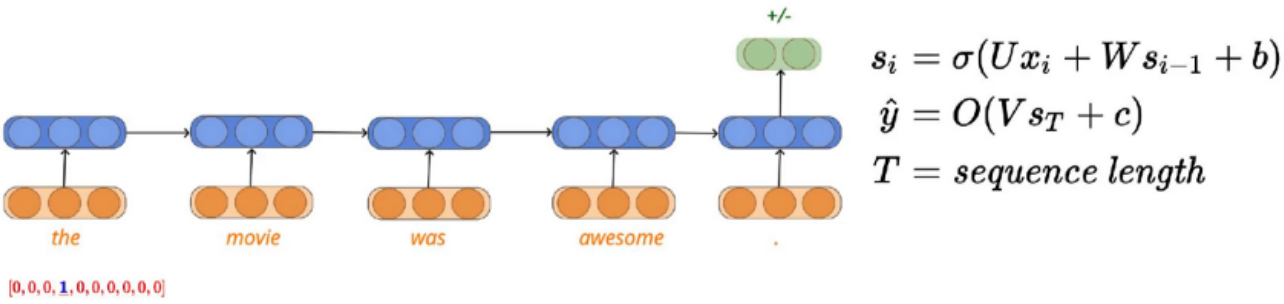
Model

Model is an approximation of the true relationship that exists between the input and the output. In Sequence Learning Problems, we don't have a single input, we have a series of inputs



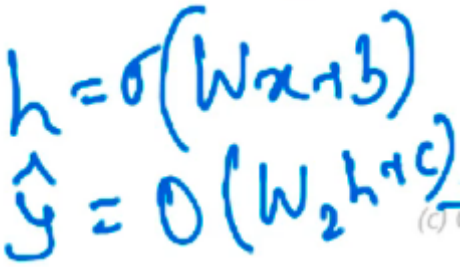
And in sequence classification problem, the output depends on the entire sequence.

So, our objective is to come up with a function such that the final output is a function of all the inputs that we have



What we have in FCNN is that output depends on a single input something like the below

And we compute the y_hat in case of FCNN as

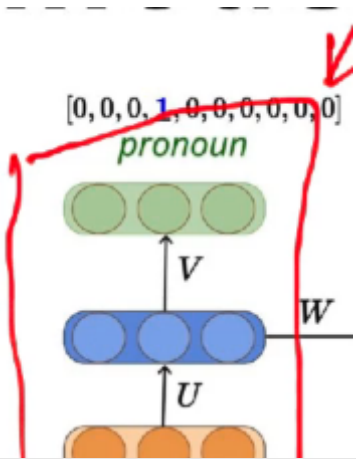


Equation of RNN is very similar to the equation in FCNN, and the only change in the equation of RNN is that instead of depending only on the input, hidden state(si) depends on the previous hidden state and the final y_hat(was computed from single input in case of FCNN) here we compute from multiple inputs, we keep on accumulating the hidden states(s1, s2, s3, ….., sT) and then using sT we compute the final output. This is going to be a long computation as there are so many non-linearities, first we are computing sigmoid over x1 then in the next step we are computing sigmoid over s1 and x2 after multiplying both with weight matrix and so on.

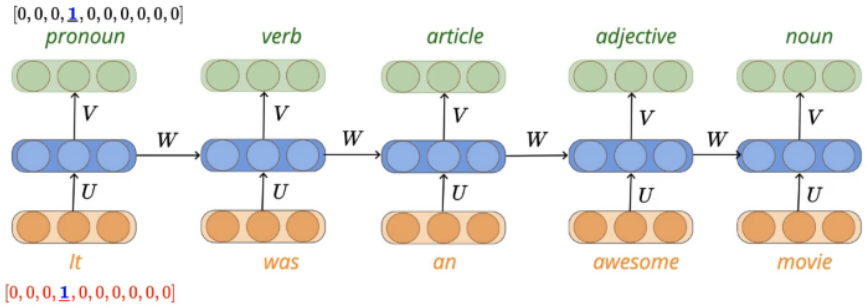
Sequence Labeling:

We have to compute output at every time step

In case of FCNN, it would have been like:

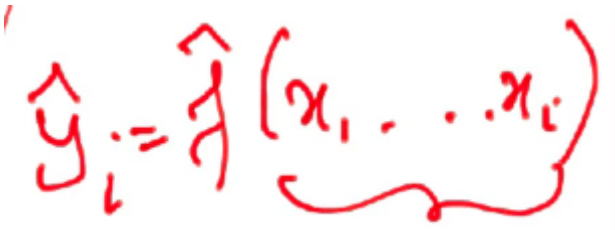


Given an input, we are computing the output which gives the probability distribution. But now this is a RNN, the subsequent output depends on previous inputs also, so we use the recurrent equation(in image below) again and here the idea also remains the same, instead of the final output y_hat, now we are interested in output at every time step and that depends on current hidden state and this hidden state depends upon previous hidden state as well as the current input. So, we keep applying this equation again and again and at each time step we can compute the probability distribution and it would give the output.



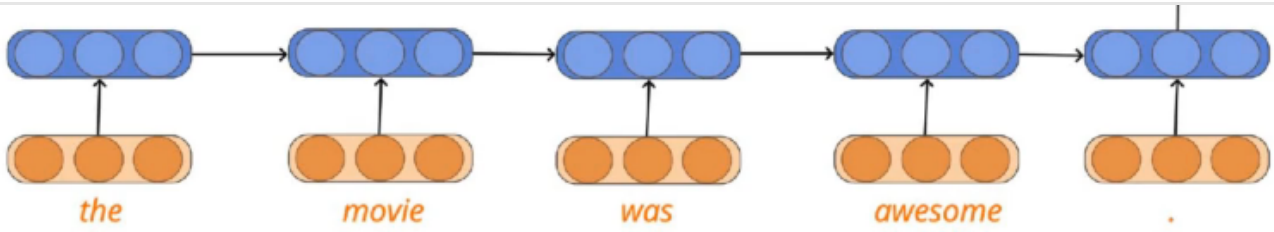
$$s_i = \sigma(Ux_i + Ws_{i-1} + b)$$
$$\hat{y}_i = O(Vs_i + c)$$

So, we have y1_hat, y2_hat and so on and that depends on current hidden state and that hidden state depends upon previous hidden state as well as the current input



Loss Function:

Sequence Classification Task:



Let's say there are 2 classes positive and negative and in this the actual label happens to be positive that means entire probability mass is on the positive class/label.

$$y = [1, 0]$$

Now to compute y_hat, we have the following equation

$$\hat{y} = O(Vs_T + c)$$

Let's say we get the y_hat as:

$$\hat{y} = [0.7, 0.3]$$

As we dealing with two probability distributions, we will use the Cross Entropy Loss which is the following:

$$\mathcal{L}(\theta) = -\sum_{i=0}^{1} y[i] \log \hat{y}[i]$$

$$= -\log y_0$$

Please note that it would be ‘y_hat’ instead of ‘y’.

So, it is negative summation over all possible values that random variable can take(in this case the possible values are 0 and 1), then the product of the true probability and the log of the predicted probability and the true output has a peculiar form where one of the entries is 0, so only one term in the summation remains and that term corresponds to the **true class**. This is for one training example. So, our total loss would be the average of the this quantity over all the training examples:

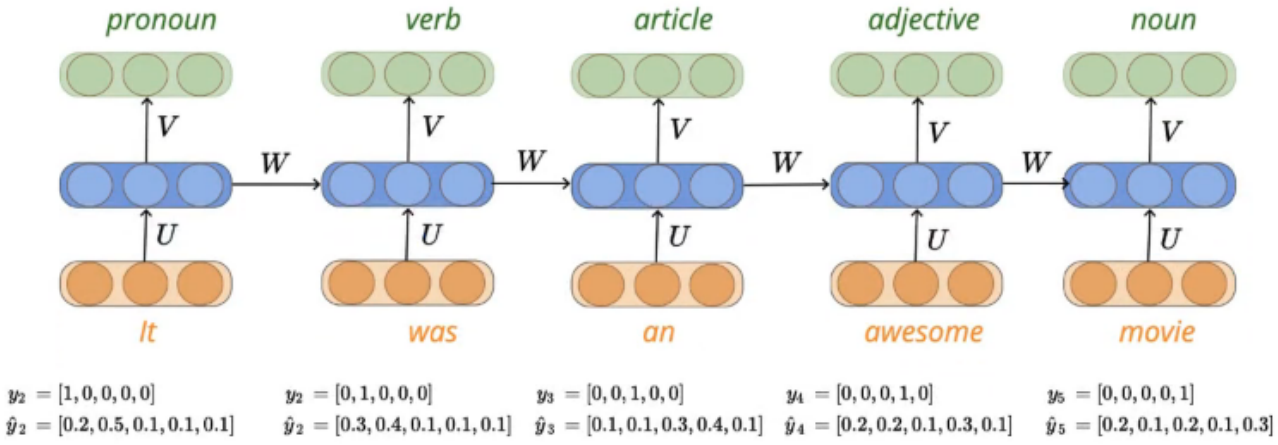
$$\mathscr{L}(\theta) = -\frac{1}{m}\sum_{i=1}^{m} \log y_{ic}$$

Please note that it would be ‘y_hat’ instead of ‘y’.

In the above image, ‘i’ is the index for the training example and ‘c’ in ‘y_ic’ is the corresponding index of the correct class.

Sequence Labeling Task:

Here at every time step we are going to make a prediction that means for every time step, we have a true distribution and a predicted distribution



$$\mathscr{L}(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{j=1}^{T}\log y_{ijc}$$

It would be y_hat(not y) in the above equation

$$\mathscr{L}(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{j=1}^{T}\log \hat{y}_{ijc}$$

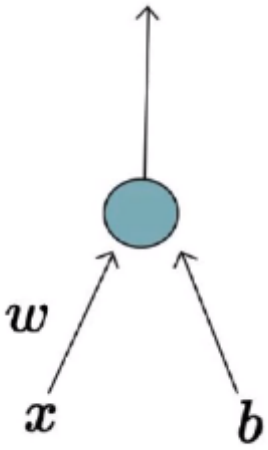
For the ‘ith’ training example, for the ‘jth’ time step, whatever is the true class the predicted probability of that then take the average of that.

Think of it like for every training example, we are making T predictions and now we have to sum up the loss that we have in all these T predictions and each of these losses(individual losses) is going to be the cross entropy or the negative log likelihood for the corresponding time step.

$$\mathscr{L}(\theta) = -\frac{1}{Tm}\sum_{i=1}^{m}\sum_{j=1}^{T}\log y_{ijc}$$

Learning Algorithm

The usual process for Logistic Regression is the following:



$$\hat{y} = \sigma(wx + b)$$

Initialise w, b

Iterate over data:

$compute\ \hat{y}$

$compute\ \mathcal{L}(w, b)$

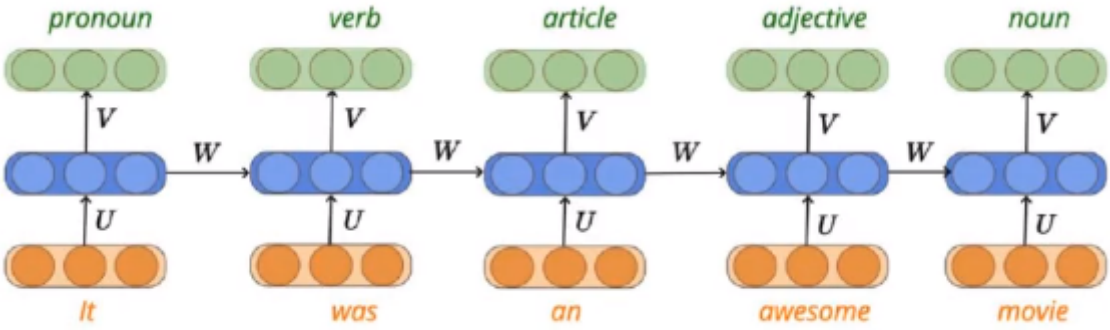
$$w = w - \eta \Delta w$$

$$b = b - \eta \Delta b$$

till satisfied

$$Earlier : w, b$$

$$Now : w_{11}, w_{12}, \ldots, u_{11}, u_{12}, \ldots, v_{11}, v_{12}$$



In this Recurrent network, we have many parameters, we have W which consists of w11, w12, … all the way up to the size of the w, similarly we have U, V, b, c(bias terms). So, the recipe is going to be the same as in case of logistic regression, instead of updating two parameters(in Logistic Regression), we update all the parameters.

Earlier we had the loss function as a function of W, b; now we will have loss function as a function of W, U, V, b, c

$$Earlier : L(w, b)$$

$$Now : L(W, U, V, b, c)$$

Total No. of Parameters:

Let's start with U

Input is of dimension 'n X 1', **U** takes this n dimensional input and convert it to 'd' dimensional output, so the dimension of **U** is going to be 'd X n'

$h \times 1$

Input dimension

Dimension of U

W takes this ‘d’ dimension vector and gives back a ‘d’ dimension vector(output of recurrent connection) that means it takes ‘d x 1’ input and gives the output of dimension ‘d X 1’ so the size of **W** is going to be ‘d X d’

Dimension of W

Now if there are ‘k’ classes, then **V** takes ‘d X 1’ input and gives back a ‘k X 1’ output, so dimension of **V** would be ‘k X d’:

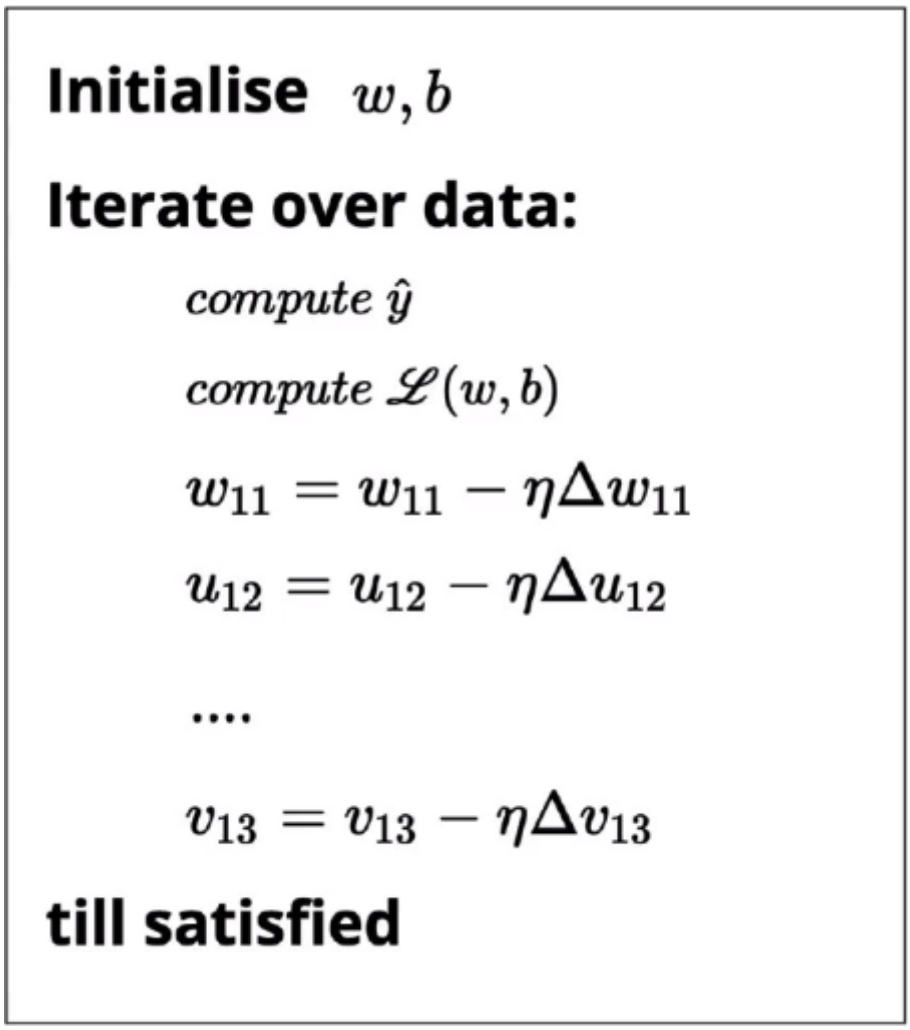
Dimension of V

b is going to be the same size as the hidden vector i.e ‘d X 1’

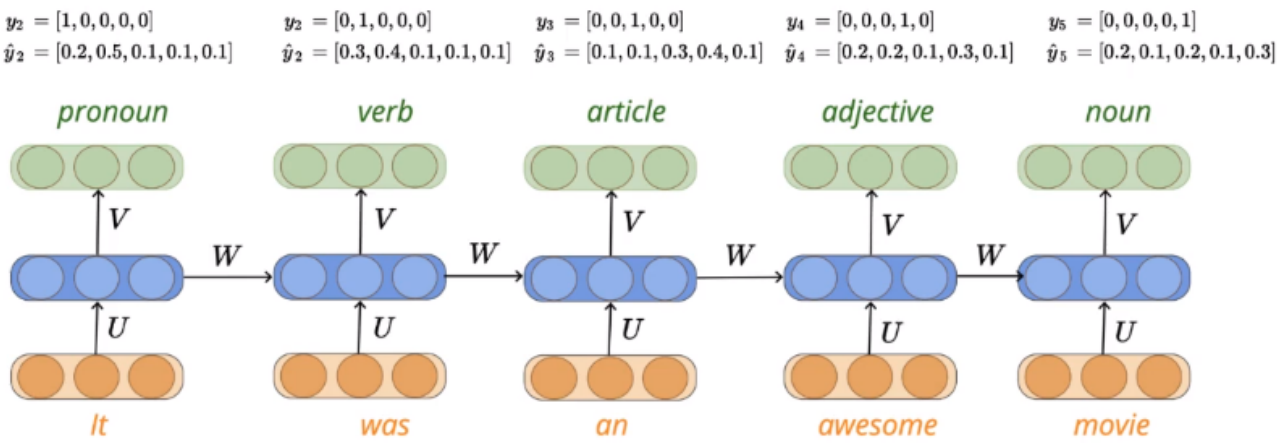
c is going to be the same size as the number of classes that we have so its dimension would be ‘k X 1’

So, the total number of parameters = (d X n + d X d + k X d + d X 1 + k X 1)

In the forward pass, we compute the s[, (y_hat)] and then we compute the loss and based on that we update all these parameters.

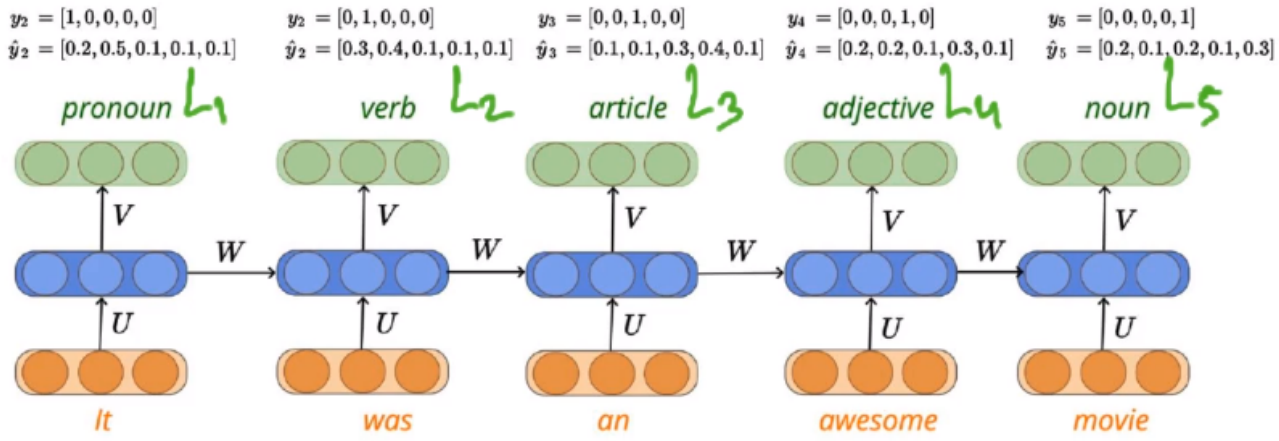


Learning Algorithm — Derivatives w.r.t. V



$$m \sum_{i=1} \sum_{j=1}$$

The loss function is just the average over all the training instance where for each training instance we accumulate the Loss for all the time steps:

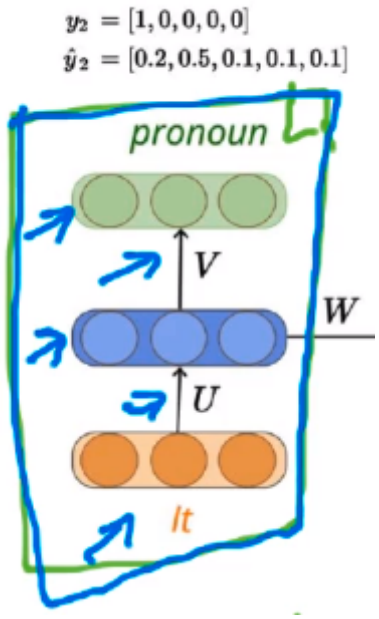


Let's say we have loss values as L1, L2, … , L5 at 1st time step, 2nd time step and so on till 5th time step in the above case/figure so the total loss **L** would be equal to (L1 + L2 + L3 + L4 + L5)

So, we have the derivative of Loss(**L**) with respect to **V** as the summation of derivative of individual loss(at each time step_ with respect to **V**

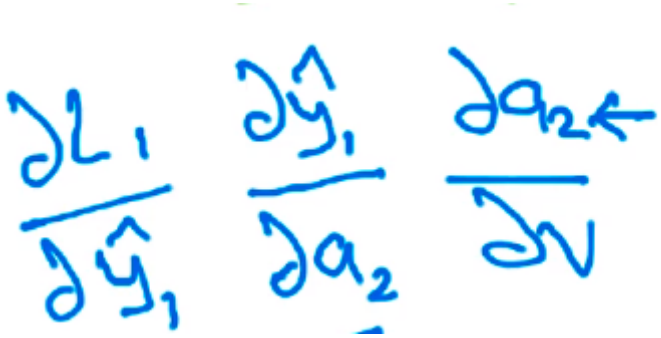
$$\frac{\partial L}{\partial V} = \frac{\partial L_1}{\partial V} + \frac{\partial L_2}{\partial V} + \cdots \frac{\partial L_5}{\partial V}$$

Now loss L1 would be computed from the following block(first block)



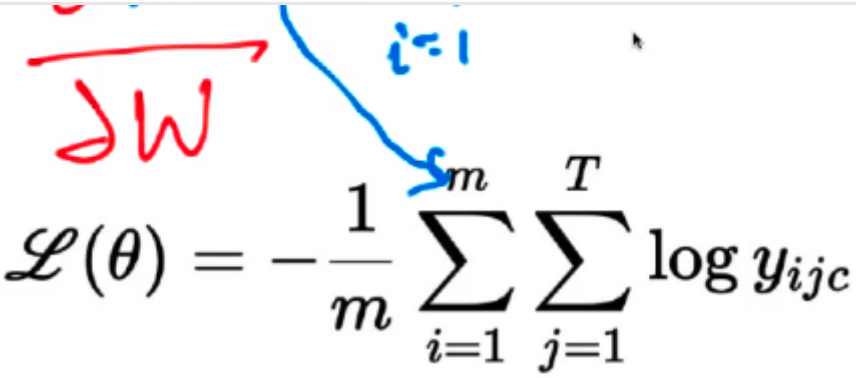
which is exactly the like a feed-forward neural network so the same formula applies here as well.

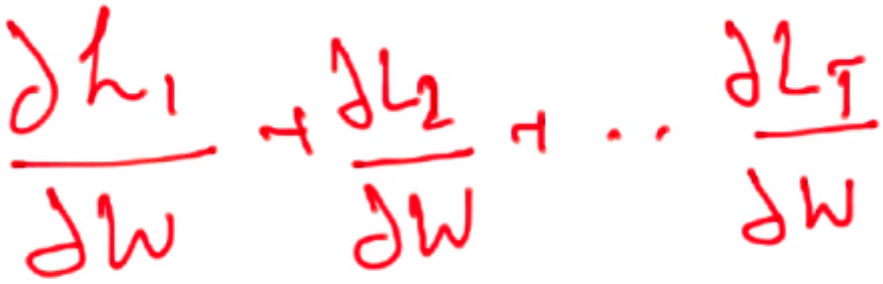
We can compute all these loss terms(L1, L2, …, L5) independently and for each term, we would have this kind of individual block(as in above picture) which is exactly like an independent feed forward network.

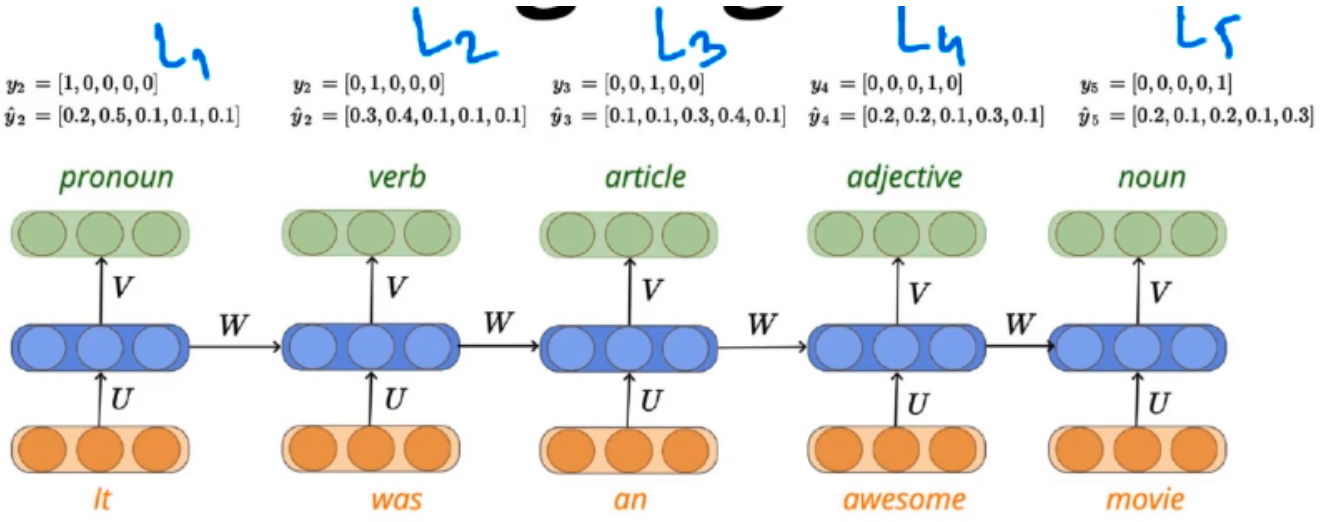


a2 is the above image refers to the pre-activation at layer 2.

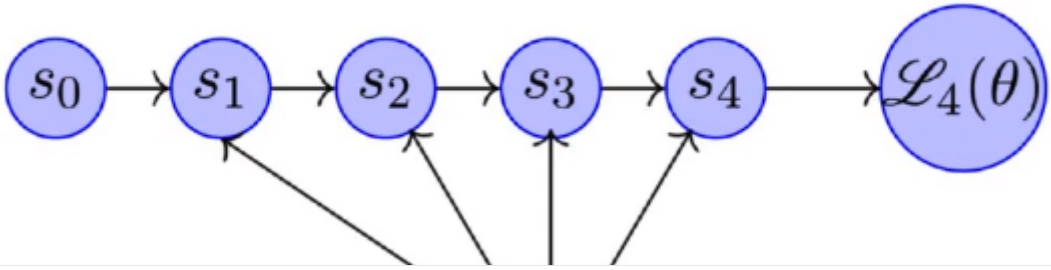
Learning Algorithm — Derivatives w.r.t. W







Let’say we want to compute the derivative of L4 wrt W

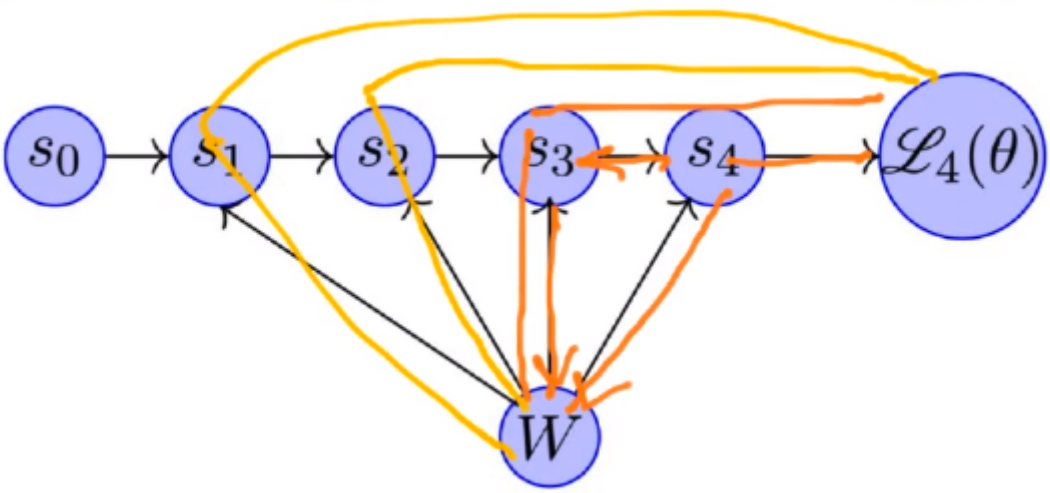


We compute the derivative of L4 wrt s4 and then the derivative of s4 wrt W

Now s4 clearly depends on W

$$s_4 = \sigma(Ux_4 + Ws_3 + b)$$

but it also depends on s3 which in turn itself depends on W and s2 and now this s2 also depends on W and this continues like this.

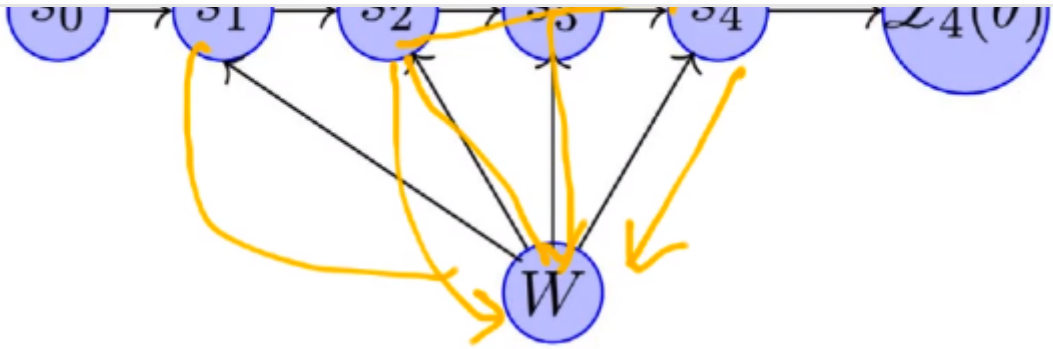


Derivative of L4 wrt W would be the sum of all the paths that leads from L4 to W

$$\frac{\partial \mathcal{L}_4(\theta)}{\partial W} = \frac{\partial \mathcal{L}_4(\theta)}{\partial s_4}\frac{\partial s_4}{\partial W}$$

Now from s4 we have multiple paths that leads to W so we need to consider all these paths

$$\frac{\partial s_4}{\partial W} = \frac{\partial s_4}{\partial W} + \frac{\partial s_4}{\partial s_3}\frac{\partial s_3}{\partial W} + \frac{\partial s_4}{\partial s_3}\frac{\partial s_3}{\partial s_2}\frac{\partial s_2}{\partial W} + \frac{\partial s_4}{\partial s_3}\frac{\partial s_3}{\partial s_2}\frac{\partial s_2}{\partial s_1}\frac{\partial s_1}{\partial W}$$



We will rewrite this equation after short-circuiting as the following:

$$\frac{\partial s_4}{\partial W} = \frac{\partial s_4}{\partial s_4}\frac{\partial s_4}{\partial W} + \frac{\partial s_4}{\partial s_3}\frac{\partial s_3}{\partial W} + \frac{\partial s_4}{\partial s_2}\frac{\partial s_2}{\partial W} + \frac{\partial s_4}{\partial s_1}\frac{\partial s_1}{\partial W}$$

which we can write as:

$$\frac{\partial s_4}{\partial W} = \sum_{k=1}^{4} \frac{\partial s_4}{\partial s_k}\frac{\partial s_k}{\partial W}$$

here 4 represent the number of time steps

Now the derivative of st(t time step) wrt sk would be given as

$$\frac{\partial s_t}{\partial s_k} = \frac{\partial s_t}{\partial s_{t-1}}\frac{\partial s_{t-1}}{\partial s_{t-2}}\cdots\frac{\partial s_{k+1}}{\partial s_k}$$

So, the overall loss wrt W is given as:

$$\frac{\partial \mathcal{L}_4(\theta)}{\partial W} = \frac{\partial \mathcal{L}_4(\theta)}{\partial s_4}\sum_{k=1}^{4}\frac{\partial s_4}{\partial s_k}\frac{\partial s_k}{\partial W}$$

And in general, we can write like the derivative of t'th time step wrt W by the following formula

$$\frac{\partial \mathscr{L}_t(\theta)}{\partial W} = \frac{\partial \mathscr{L}_t(\theta)}{\partial s_t} \sum_{k=1}^{t} \frac{\partial s_t}{\partial s_k} \frac{\partial s_k}{\partial W}$$

And this algorithm is termed as **Back propagation through time**(as we sum up the loss across all the time steps).

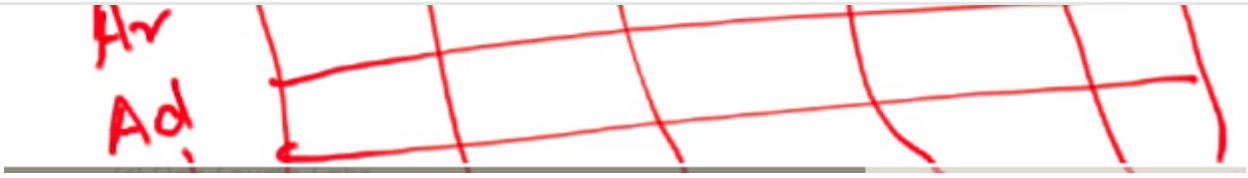
Evaluation

For sequence classification tasks we have the simple accuracy matrix and is computed as the No. of correct predictions divided by the total No. of training examples.

For multi-class classification(like sequence labeling tasks), we can still compute the overall accuracy as: we compute the class for each word in every sentence and from this, we count the number of correct predictions(words which were predicted correctly) then we divide this by the total number of words

We can also compute the accuracy per class for ex. let's say there are 5 classes, Pronoun, Article, Verb, Adjective, Noun and we want to compute the accuracy for the Pronoun, let's say out of total number of words that we have, 5 words are actually Pronoun and our model predicts 3 of them as the noun so our accuracy in this case would be (3 / 5) = 60%.

We can also make a confusion matrix which gives complete picture across all the classes.



Here the entry in the Horizontal represents the true class for example in the first row, P represents that the true label/class is Pronoun and then the number in front of that row represents the number of predictions across all the classes(class name in vertical) for the words which are actually Pronoun; from the above figure we can say that out of the total words that were pronoun, model predicted 2 words as Verb, 1 as Article and so on.

This matrix tells us where the confusion is, where the model is facing an issue, which are the classes getting confused.

For an ideal confusion matrix, numbers across the diagonals must be maximum.

All the images used in this article is taken from the content covered in the RNN module of the Deep Learning Course on the site: padhai.onefourthlabs.in

Machine Learning Rnn Padhai Deep Learning Recurrent Neural Network